VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Podpora testování sekvenčních kódů v nástroji Kaira

Support for Testing of Sequential Codes in the Tool Kaira

2015 Pavel Siemko

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Pavel Siemko**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Podpora testování sekvenčních kódů v nástroji Kaira
Support for Testing of Sequential Codes in the Tool Kaira

Zásady pro vypracování:

Cílem práce je rozšířit nástroj Kaira o podporu testování sekvenčních kódu. Kaira je vývojové prostředí, které slouží k vytváření paralelních aplikací určených pro systémy s distribuovanou pamětí. Programovaní v Kaiře kombinuje principy, tzv. vizuálního programování s obvyklým (textovým) popisem programu. Vizuální jazyk, který je inspirován Barevnými Petriho sítěmi slouží pro popis komunikace v programu, zatímco výkoné části jsou psány standardně v C/C++.
V současné době je možné testovat tyto sekvenční části programu pouze za pomocí externích nástrojů. S tím, že je zde reálný problém s částmi psanými v C/C++, jelikož se typicky jedná o pouhé fragmenty kódu, které jsou napojeny na vizuální části a není možné je snadno přenést do onoho externího nástroje. Hlavním cílem práce je tedy umožnit uživateli vygenerovat samostatný test pro konkrétní fragment kódu, který bude nezávislý na zbytku programu a bude jej možné snadno přenést do externího nástroje.

1. Seznamte se s nástrojem Kaira a prozkoumejte jeho možnostmi.
2. Navrhněte a rozeberte možná řešení, s tím, že se soustřeďte na části kódu umístěných uvnitř přechodů. Berte v úvahu stav výpočtu a data vstupující do přechodů.
3. Implementujte prototyp vámi zvoleného řešení.
4. Podívejte se na možnosti testování C/C++ výrazů na hranách a zbývajících komponentách sítě.

Seznam doporučené odborné literatury:

[1] Stanislav Böhm: Unifying Framework for Development of Message-Passing Applications, http://verif.cs.vsb.cz/sb/thesis.pdf

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Šurkovský**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka	prof. RNDr. Václav Snášel, CSc.
vedoucí katedry	*děkan fakulty*

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. června 2015 .....

Rád bych touto formou poděkoval zejména mé přítelkyni Lence Klapuchové, která mě podporovala, dělala jazykovou korekturu a věřila ve mě, dále Mgr. Bronislavu Klapuchovi, za odbornou pomoc při psaní textu bakalářské práce, mému vedoucímu bakalářské práce Ing. Martinu Šurkovskému, který mi pomohl se samotnou funkčností a v poslední řadě autorovi nástroje Kaira Ing. Stanislavu Böhmovi Ph.D., který si na mě udělal čas a vysvětlil mi funkčnost tohoto nástroje, abych mohl tuto práci dokončit.

Abstrakt

Bakalářská práce uvádí do problematiky testování softwaru a její hlavní součástí je nově vytvořená funkcionalita nástroje, který dokáže pracovat v rámci testování sekvenčního kódu a přináší tak nové možnosti rozšíření techniky testování. Práce se zaměřuje na podporu testování sekvenčního kódu a tvorbu jednoduchých unit testů. Čtenářům přiblíží samotné testování, debugování, verifikaci a paralelní systémy. Dále objasní rozšiřovaný nástroj a rozebere možnosti řešení daného problému a uvede jedno z řešení, které bylo implementováno. Hlavním přínosem této práce je vytvořená funkcionalita pro umožnění testování jednotlivých přechodů v testovacím prostředí a generování nové zjednodušené sítě. Z takto zjednodušené sítě je následně generován kratší (jednodušší) kód, což umožní přehlednější práci s kódem v libovolném externím IDE, např. pro účely debuggování.

Klíčová slova: Kaira, MPI, testování, debuggování

Abstract

The Bachelor thesis gives an insight into the issue of software testing. The main part of the thesis is a new functionality of a tool which is able to work in the sequentional code testing frame. It brings us this way the new extended testing possibilities. The thesis is focused on the support of the sequentional code testing and the simple unit-tests development. It brings the reader into testing area, debugging, verification, and parallel systems. Further, it illustrates a peddled tool and looks at options to solve the problem and gives one of the solutions, which have been implemented. The main benefit of this thesis is the fully developped functionality for giving us the posibility of the interfaces testing in test environment and generation of the new simplified network. From these simplified network is subsequently generated less (simpler) code, allowing clearer work with the code in any external IDE, e.g. for debugging purposes.

Keywords: Kaira, MPI, testing, debugging

Seznam použitých zkratek a symbolů

MPI	–	Message Passing Interface
OS	–	Operační systém
Id	–	Identifikační číslo

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Pro svou bakalářskou práci jsem si zvolil vytvoření vlastní nové funkcionality pro nástroj Kaira, která umožňuje testování sekvenčního kódu v rámci průběhu přechodu. Použití nástroje je velmi dobře využitelné v aplikaci paralelních aplikací s distribuovanou pamětí. Nástroj se jmenuje Kaira. Programování zde kombinuje principy tzv. vizuálního programování s běžným kódováním. Vizuální část je inspirována barevnými Petriho sítěmi a slouží k popisu komunikace mezi procesy v programu, zatímco výkonná část je napsána v jazyce C/C++.

Testování sekvenčního kódu v síti bylo dosud možné pouze za pomoci použití externích nástrojů, vygenerováním kódu pro celou síť, ne jen pro jeden přechod, což bylo zdlouhavé a pro testování nedostatečně intuitivní. Hlavním problémem však byla nemožnost úplného komplexního přenosu vstupních informací do onoho externího nástroje.

Nově vytvořená funkcionalita umožňuje generovat menší nezávislé sítě, ze kterých poté uživatel vytváří vlastní testy. Generování probíhá buď do stejného projektu jako podsíť, nebo do projektu nového. Z dané sítě poté uživatel vytváří vlastní jednotkové testy, které může automaticky kompilovat a poté spouštět v testovacím prostředí. Všechny generované sítě se automaticky přidávají k projektu jako test. Pokud chce uživatel přidat vlastní test, který není generovaný novou funkcionalitou, může tak učinit v testovacím prostředí. Nově vytvořená funkcionalita umožňuje dále ukládat vstupní hodnoty přechodu, které se promítají do všech vygenerovaných projektů. Vše potřebné je v práci popsáno a zároveň podloženo přiloženým programem.

Ve druhé kapitole je definováno testování a chyby v programu. Následně je testování rozděleno do několika kategorii. Protože se práce zaměřuje na generování automatických jednotkových testů, jsou v práci uvedeny nejpoužívanější testovací nástroje, které jsou založeny na nejpoužívanější architektuře testovacích frameworků, xUnit. Na konci kapitoly je čtenář seznámen s jinými způsoby hledání chyb. Ladění programu, které se využívá více k řešení chyb, než ke hledání a verifikace, která hledá chyby z pohledu matematických důkazů.

Ve třetí kapitole jsou uvedeny pojmy proces, vlákno a paralelismus. Paralelní systémy jsou pote rozděleny do kategorii a je přiblížena architektura pamětí, které jsou používány v paralelních systémech. Také je uveden a rozveden pojem MPI a přiblížen samotný nástroj Kaira.

Ve čtvrté kapitole je přiblíženo samotné zadání. Jsou uvedeny dvě možnosti řešení, jejich výhody a nevýhody.

V páté kapitole je popsána implementace prototypu jednoho z řešení krok po kroku. Je také uveden manuál použití nové funkcionality.

Šestá kapitola popisuje poslední část zadání, a to možnost, jak by se daly testovat fragmenty kódu na hranách a inicializační kód míst.

Sedmá kapitola popisuje zhodnocení a závěr práce, ve kterém je shrnutí celé práce. V možnostech rozšíření jsou kromě věcí, kterými lze navázat na tuhle práci, uvedeny problémy, které se mi nepodařily vyřešit.

2 Testování

Testování je nedílnou součásti vývoje softwaru. Testování slouží k odhalení chyb v softwaru. Testování lze popsat jako proces získávání informací o stavu a vlastnostech systému za účelem jejich dalšího zpracování[2].

2.1 Chyby v programu

Programátorská chyba je něco, co může způsobit, že funkce neplní požadovanou funkčnost.

Programátorské chyby dělíme na syntaktické a sémantické. Syntaktická chyba představuje prohřešky vůči gramatice používaného programovacího jazyka. Takové chyby ale překladač vždy odhalí. Druhým druhem chyb jsou sémantické chyby. Sémantická chyba značí prohřešek proti požadované logice programu. Všechny tyto chyby nelze detekovat automaticky, proto způsobují chybný běh programu a na jejich odhalení se zaměřuje právě testování a ladění (debugování) programu. Sémantické chyby lze odhalit testováním, laděním, nebo verifikací. Zvláštním případem chyb v programu jsou vnější chyby. Vnější chyby jsou ty, které jsou způsobeny hardwarem, nebo dokonce softwarem. Příkladem této chyby je například to, že operační systém nepřidělí naši aplikaci dostatek volné paměti, nebo software, se kterým pracuje náš systém najednou nebude fungovat jak má. Za tyto chyby nenese programátor přímou zodpovědnost[5].

2.2 Rozdělení testování

Testování se dělí do několika kategorií. V těchto kategoriích se používají různé kombinace druhů a typů testů. Všechny kategorie ale mají podobný průběh testování. Ten se dá rozdělit na 3 části:

- Plánování testů - Definice projektu, v jakých iteracích a jak testovat a vyhodnocovat.
- Analýza a příprava testů - Návrh jednotlivých testovacích případů a příprava testovacích dat.
- Provedení a vyhodnocení testů - Provedení testů a zaznamenání jejich výsledků. Analýza a vyhodnocení, zda jsme dosáhli očekávaného výsledku.

Testování lze dále rozdělit podle různých ukazatelů: jaké jsou požadavky na kvalitu kontroly programového kódu, jakým způsobem se testy provádějí, rozměr kvality, které testy ověřují a v jaké fázi vývoje se vývoj softwaru nachází[2, 3].

2.2.1 Míra znalosti testovaného kódu

Míra znalosti testovaného kódu se dá charakterizovat jedním z těchto dvou pojmů *testování černé skříňky* a *testování bílé skříňky*. Při testování černé skříňky tester neví, jaká je logika daného kódu a nemůže se na ni podívat. Má k dispozici pouze definované vstupy a jim odpovídající definované výstupy.

Při testování bíle skřínky tester zná logiku daného kódu, může nahlédnout přímo do zdrojového kódu programu, zná jeho datový model a strukturu. Jeho zkoumání mu tak může napomoci při tvoření a vyhodnocování testů.

2.2.2 Statické vs. Dynamické testování

Testy dělíme také na statické a dynamické. Pro statické testy není potřeba spouštět aplikaci. Testuje se zde to, co neběží - to znamená, že zkoumaný objekt prohlížíme a kontrolujeme. Mezi statické testování se dále řadí kontrola zdrojového kódu. Daný kód může být předán někomu dalšímu, který kód kontroluje a hledá v něm chyby ještě dříve, než se daný kód spustí. Dynamické testování je pravým opakem. Testujeme tak, že software spustíme a pracujeme s ním (testujeme ho).

Mezi statické testování lze také zařadit testování specifikací softwaru. Specifikaci si lze představit jako dokument, jež vznikl složením několika studií z různých zdrojů - studie použitelnosti, marketingové studie atd. Zkoumá se, zda specifikace neobsahují možné chyby, které v pozdější fázi vývoje mohou určit špatný směr.

2.2.3 Funkční vs. Nefunkční testy

Funkčními testy označujeme takové testy, které nějakým způsobem kontrolují samotnou funkčnost produktu. Kontrolují se zde všechny části a funkce produktu tak, aby odpovídaly požadavkům zadání, aby neobsahovaly chyby atd.

Nefunkčními testy rozumíme takové testy, které netestují funkce a vlastnosti produktu. Řadíme zde takové testy, které ověřují např., jak se bude produkt chovat při různých konfiguracích, při výkonnostním testování, nebo jak se bude produkt chovat pod větší zátěží, zda bude dostatečně rychlý a s malou odezvou (větší množství uživatelů pracujících v produktu v jednu chvíli). [4]

2.2.4 Manuální vs. Automatické testy

Testy lze dále rozdělit na testy, které provádí člověk a na testy, které jsou prováděny prostřednictvím speciálních softwarových nástrojů. U manuálních testů testování provádí a vyhodnocuje člověk. Provádí testování podle jednotlivých kroků popsaných v testovacích scénářích, a poté zaznamenává výsledek. Nevýhodou těchto testů je to, že jeden test může trvat několikanásobně déle, než test, který je automatizován. Na druhou stranu může lépe reagovat na případné neočekávané změny.

Automatizované testování provádí speciální nástroje, nebo speciální jazyky, pomocí kterých jsou tvořeny testovací případy, které buď rozšiřují funkčnost základních programovacích jazyků, nebo se úplně liší. Daný test je napsán jako skript do automatizovaného nástroje, ale také existují případy, kdy je test psán pomocí kompilovaného kódu. Lze zde provádět velké množství testů v krátkém čase. Příkladem tohoto druhu testů jsou např. selenium testy (testují funkčnost webových formulářů), nebo jednotkové (unit) testy. Příkladem rozšíření programovacího jazyka je testovací framework JUnit, který rozšiřuje funkčnost jazyka Java. Využívá architektury xUnit pro jednotkové testy.

Dále se budeme spíše zaměřovat na automatické jednotkové testy.

2.3 Jednotkové testy

Jednotkový test je speciální případ automatických testů. Jednotkové testy píše vývojář a používají se k ověřování některých funkcí a oblastí, nebo jednotek kódu v průběhu vývoje softwaru. To znamená, že pro každou funkci a množinu vstupních hodnot se může určit, zda funkce vrací správné hodnoty pro zadané vstupní, nebo mezní hodnoty a havaruje, nebo indikuje výjimku pro neplatný vstup. Tohle vše pomáhá identifikovat nedostatky v algoritmech, nebo samotné logice.

2.4 Testovací nástroje

Ve všech moderních programovacích jazycích existuje nástroj, který umožní jednotkové automatizované testování. Příkladem takového nástroje jsou např. JUnit, PyTest, NUnit nebo Google C++ Testing framework. Všechny tyto testovací nástroje využívají architekturu xUnit.

2.4.1 xUnit

xUnit je název pro skupinu testovacích frameworků založených na stejné architektuře. Jméno je odvozeno od prvního z nich, JUnit. Původ tohoto frameworku začal jazykem Smalltalk. Kent Beck byl velký fanoušek automatizovaných testů v období vývoje a tak vymyslel jednoduchý framework pro chod jednotkových testů. Důraz byl kladen na to, aby mohl programátor snadno definovat test za použití jeho vývojového prostředí pro Smalltalk, a poté je mohl snadno a rychle pouštět po jednom, nebo všechny naráz[9].

Testovací nástroje, z této rodiny frameworků, nabízí základní komponenty architektury, s různými druhy implementace[10].

- Test runner - program, který spouští jednotlivé testy implementované pomocí frameworku z rodiny xUnit a vykazuje výsledky jednotlivých testů.
- Test case - Základní třída, z které dědí všechny jednotkové testy.
- Test fixtures - je to množina podmínek nebo stavů potřebných ke spuštění testu. Programátor by měl vytvořit funkční stav aplikace před testem a po ukončení testu zase navrátit stav aplikace do původního stavu.
- Test suite - sada testů, jenž sdílejí stejné příslušenství. Nezáleží na pořadí provádění testů.
- Test result - sběr výsledků testovacích funkcí.
- Assertion - funkce, metoda nebo makro, které ověřuje stav nebo chování jednotky v rámci testu. Selhání tvrzení obvykle vyvolá výjimku a zastaví vykonávání aktuálního testu.

Nová funkcionalita obsahuje některé z těchto znaků xUnit architektury. Jsou jimi například Test runner, Test result nebo Assertion.

Porovnání nejpoužívanějších testovacích nástrojů z rodiny xUnit pro některé z nejznámějších programovacích jazyků se nachází níže.

- JUnit[1] - je testovací nástroj pro programovací jazyk Java. Testy se zde píší do tříd, které jsou umísťovány do speciálních složek. Jednotlivé testy se zde označují anotacemi (`@Test`) a jeden test tvoří jedna metoda třídy. V anotaci lze určit, která výjimka se očekává, že bude vyvolána `@Test(Expected=IOException.class)`.
- NUnit[2] - je testovací nástroj pro programovací jazyky z rodiny .NET. Testy se zde také píši do tříd (třída musí být označena atributem `[TestCase]`), kdy jeden test reprezentuje jedna metoda označená atributem `[Test]`. Lze také určit zda se očekává vyvolání výjimky pomocí `[ExpectedException( typeof( Argument-Exception ) )]`
- Google C++ Testing Framework[3] - Testovací nástroj pro jazyk C++ vyvinutý firmou Google. Testovací funkce se zde píší jako makra. Každé makro má dva parametry: test case name a test name. Testovací soubor má příponu *.cc* a spouští se celý soubor naráz. Funkce assertion (tvrzení) zde existují.
- PyTest[4] - Je testovací framework pro programovací jazyk Python. Jednotlivé testy se do souboru píšou jako globální funkce. Testy se poté spouští jako jeden soubor.

Všechny tyto nástroje mají podobné ovládání. JUnit a NUnit mají hromadné spouštění testů, kontrolu výsledků testů, spouštění pouze neúspěšných testů atd. zabudované přímo ve vývojovém prostředí jazyka (např. Visual Studio, NetBeans). Zato testovací nástroj pro C++ a Python spouštějí testy přes samotné spuštění testovaného souboru s testy v příkazovém řádku, nebo z jiného spustitelného souboru.

2.5 Další možnosti hledání chyb

Kromě testování existují ještě dvě možnosti, jak nalézt chyby. Jeden způsob je matematického původu - verifikace. Druhý způsob se používá převážně k řešení chyb - debuggování. Uživatel už přibližně ví, kde se nachází chyba, ale krokováním jednotlivých řádků kódu se snaží nalézt přesná příčina.

2.5.1 Ladění programu

Laděním obvykle rozumíme systematické hledání chyb pomocí krokování v debuggeru. Také se ale používá k pochopení, co daný kód dělá. Debugger je program, který pozmění

[1] http://junit.org/
[2] http://www.nunit.org/
[3] http://code.google.com/p/googletest/
[4] http://pytest.org/latest/

daný kód programu tak, že dosadí vlastní zarážky na místa, kde je definována zarážka (breakpoint) a umožní tak pozastavit běh programu na daném příkazu, nebo umožní běh programu krokovat po příkazech, popř. řádcích zdrojového kódu. V okamžiku pozastavení programu si lze prohlížet či měnit obsah paměti konkrétních proměnných, stav zásobníků, registrů a podobně. Tímto způsobem můžeme ověřit, zda program splňuje funkčnost, která byla zamýšlena[5].

2.5.2 Formální verifikace

Stanovení vlastností softwarového nebo hardwarového návrhu používající logiku, nikoli jen testování nebo neformální argumenty. Jedná se o formální specifikaci požadavků, formální modelování implementace a přesná pravidla k odvozování, aby se prokázalo, zda daná implementace splňuje specifikaci. Formální verifikace je činnost, která potvrdí nebo vyvrátí správnost systému s ohledem na jeho vlastnosti nebo formální specifikace[8].

Při klasickém testování softwaru, není jistota, že jsou nalezeny veškeré chyby, které se v softwaru mohou nacházet. Na rozdíl od verifikace, která využívá ověřování správnosti softwaru na základě matematických formálních metod. Formální verifikace ověřuje vlastnosti systému, které musí být definovány na začátku. Prohledává celý stavový prostor a hledá takové stavy, které splňují předem stanovené podmínky.

3 Paralelní systémy

Procesem se myslí aplikace, nebo program, který zavedl do operační paměti operační systém. OS pro každý proces vytváří tzv. virtuální adresní prostor, tedy paměťový prostor, který náleží danému procesu a ve kterém se proces nachází.

Vláknem se myslí objekt OS, ve kterém běží programový kód. Vlákno je skutečnou pracovní jednotkou. Při vytvoření nového procesu vzniká proces a jeho adresní prostor a vytvoří se vlákno, ze kterého se aplikace spustí. Toto vlákno se nazývá primární a OS ho vytváří zcela automaticky.

Primární vlákno může tvořit nové vlákna. Tak samo může i každé vytvořené vlákno vytvářet další vlákna. Takových vláken může být libovolné množství. Všechna vlákna jednoho procesu využívají tentýž společný virtuální adresní prostor procesu. Adresní prostor je definován procesem, ne vláknem, proto je možné sdílet data procesu více vlákny.

Úloha (Task) je objekt, jehož kód se sekvenčně vykonává. Každému vláknu procesu odpovídá jedna úloha[7, 11].

3.1 Paralelismus

Systém, v němž může probíhat několik procesů současně, označujeme jako paralelní systém. Paralelismus má využití ve zvyšování výkonu číslicových systémů vhodným rozdělením úkolů mezi jednotlivé složky paralelního systému. Můžeme tedy provést několik operací současně, tím se zkrátí doba pro vykonání zadaného úkolu a tak dochází k růstu efektivity celého systému. Tento růst efektivity je ale omezen, jak velikostí komunikace, tak samotným počtem procesorů. Také je zde omezení ohledně rozdělení velkého problému na menší, nelze rozdělovat problém do nekonečna.

Paralelními systémy lze také chápat jako systémy, které paralelně řeší výpočetně složité problémy rozdělením na menší podproblémy, které se dále paralelně zpracovávají. Úrovně granularity paralelního procesu vyjadřují, jak velké celky se zpracovávají současně[6].

- Hrubozrnná granularita - Hovoříme zde o systémech s více než jedním procesorem, kde je paralelismus na úrovni procesů
- Jemnozrnná granularita - Hovoříme zde o paralelismu na úrovni příkazů.

Paralelní systémy se dále dělí na homogenní a heterogenní. Homogenní systémy jsou tvořeny procesory stejného typu se stejnou sadou instrukcí. Heterogenní systémy mohou tvořit různé počítače s různými procesory, s různou architekturou, s různými operačními systémy a různým zobrazením dát.

3.2 Rozdělení paralelních systémů

Výpočetní systém je zařízení, do kterého vstupuje jeden nebo více typů dat a jeden nebo více typů instrukcí, které zpracování dat řídí.

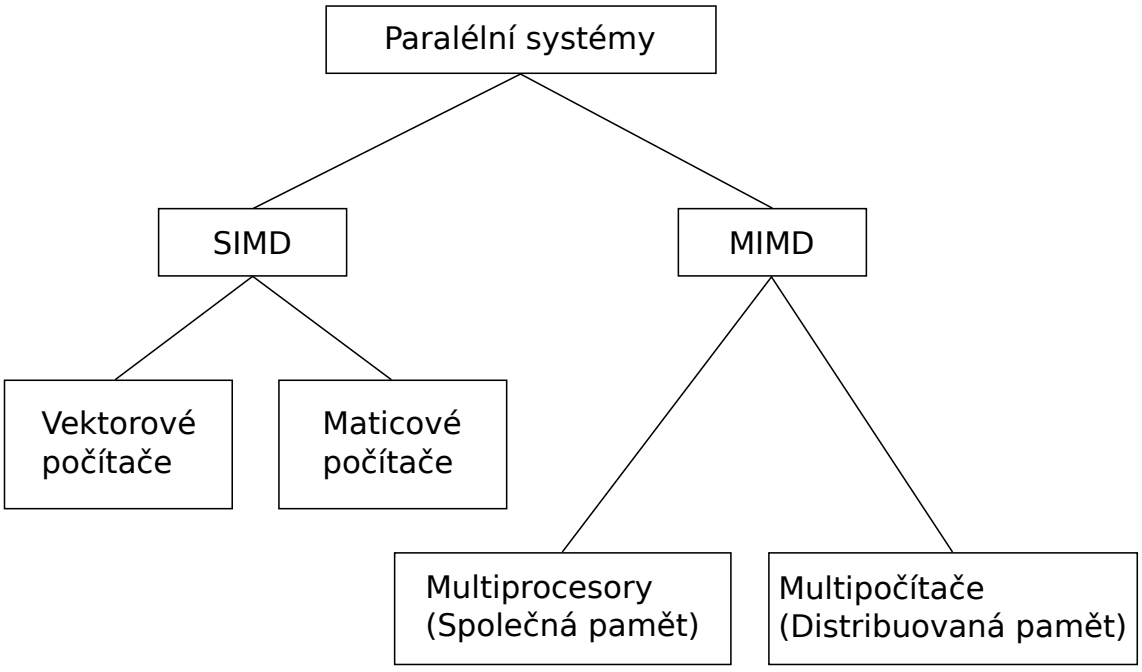
Výpočetní systémy se dělí podle charakteristických vlastností:

- SI (Sigle Instruction) - Zpracování jednoho toku instrukcí
- MI (Multiple Instruction) - Zpracování více toků instrukcí
- SD (Sigle Data) - Zpracování jednoho toku dat
- MD (Multiple Data) - Zpracování více toků dat

Dále se výpočetní systémy řadí do různých kategorií podle zpracování instrukcí a dat (viz tabulka 1). Obrázek 3.1 popisuje nejpoužívanější typy architektury paralelních systémů. Existují také SISD a MISD typy. Typ SISD je zpracovávaní dat sériově podle jedné instrukce. V dnešní době se již tato architektura nevyužívá. Příkladem tohoto typu je von Neumannův počítač. Typ MISD je zpracování jedněch dat více instrukcemy. Vyžaduje serii procesorů. Tento typ není v praxi běžný.

	SI	MI
SD	SISD	MISD
MD	SIMD	MIMD

Tabulka 1: Rozdělení výpočetních systémů podle M.J.Flynn.



Obrázek 3.1: Ukázka Flynnovy klasifikace paralelních systémů (Flynnová taxonomie).

3.3 Organizace paměti

Kromě počtu procesorů, jejich rychlosti a topologie jejich propojení, je výkon počítače ovlivněn také velikostí a rychlostí paměti.

Rychlost paměti dělíme na vybavovací dobu a cyklus paměti. Vybavovací doba je doba, ve které se vykoná jeden zápis nebo čtení z paměti. Cyklus paměti je délka intervalu mezi dvěmi po sobě jdoucími požadavky na paměť.

Pro snížení cyklů paměti se používá paměť s prokládanými cykly. Paměť je zde rozdělena do bloků, kde každý blok je nezávislý na ostatních, pracuje samostatně a je schopen provádět čtecí nebo zapisovací cyklus nezávisle na ostatních. Každý procesor poté komunikuje střídavě s různými paměťovými bloky. Díky tomu mohou jednotlivé cykly probíhat paralelně a komunikace paměti s procesorem se zrychlí[6].

3.3.1 Sdílená paměť

Bloky hlavní paměti jsou pro všechny procesory společné, tzn. paměť je sdílená pro všechny procesy. K propojení se využívá propojovacích sítí, které umožňují propojit libovolný procesor s libovolným blokem paměti. Toho se využívá v komunikaci mezi procesory, kdy spolu komunikují prostřednictvím dat, které ukládají do paměti. Proto jsou vhodné pro úlohy s velkou rychlostí komunikace a s velkými nároky na rozsah. Může nastat konflikt při komunikaci a to problém s konzistencí dat.

3.3.2 Distribuovaná paměť

Každý procesor má svůj blok paměti a také může mít i vlastní soubor periferií. Stává se tak jádrem počítačového uzlu. Všechny tyto uzly spojuje propojovací sít. Výhodou takového rozdělení procesorů a pamětí je, že nemůže nastat konflikt dat a přístup procesoru k paměti může přistupovat plnou rychlostí bez vyrušení ostatními procesory. Procesy mezi sebou komunikují formou zasílání zpráv přes propojené sítě.

3.3.3 Hybridní paměť

Hybridní architektura paměti je kombinací sdílené a distribuované paměti. Uzel je zde reprezentací více procesorů, které mají společnou sdílenou paměť. V uzlu tedy platí to, co pro architekturu sdílené paměti. Distribuovaná architektura se zde promítá v komunikaci mezi jednotlivými uzly. Aby mezi sebou mohly komunikovat, posílají si mezi sebou zprávy.

3.4 MPI - message passing interface

MPI - Message Passing Interface je knihovna specifikující rozhraní pro (explicitní) programování paralelních aplikací na jednom nebo více počítačích s distribuovanou paměti. Startuje několik procesorů, které si mezi sebou zasílají zprávy, přičemž každý má svůj vlastní blok paměti. Pokud chceme docílit maximálního výkonu a odstranění zpoždění přepínáním kontextu, přidělíme každému procesoru jeden proces. MPI je nezávislé na

programovacím jazyce, ale převážně se setkáme s implementacemi v jazycích C, C++ a Fortran. Existuje mnoho implementací této knihovny, například OpenMPI[5], MPICH[6] a LAM[7]. MPI je standardizováno na MPI fóru[8][1].

3.5 Nástroj Kaira

Nástroj Kaira[9] je open-source vývojové prostředí pro tvorbu paralelních systémů s distribuovanou pamětí pomocí rozhraní MPI. Samotný nástroj má sloužit k jednotnému prostředí pro vývoj, testování, ladění, predikci výkonu a verifikaci aplikací. Programování zde kombinuje principy tzv. vizuálního programování s obvyklým popisem části programu formou textu. Vizuální jazyk je inspirován barevnými Petriho sítěmi[10], které slouží pro popis komunikace mezi jednotlivými procesy (viz 3.2), zatímco výkonné části jsou psány pomocí programovacího jazyka C/C++.

Hlavní cílem Kairy je zjednodušit vývoj paralelních aplikací s distribuovanou pamětí. Kaira umožňuje navrhovat grafické modely paralelních systémů, vkládat sekvenční kód do přechodů modelu, generovat konečnou aplikaci, která využívá rozhraní MPI, zobrazovat simulaci, ladit samotný kód, nebo testovat sekvenční kód libovolného přechodu pomocí nově vyvinutého rozhraní pro testování.

Přechod, který má na všech svých vstupních místech v jednom okamžiku aspoň jeden token, se nazývá proveditelný. Přechod obsahuje přechodovou funkci, která se provádí při každém provedení přechodu. Tato přechodová funkce obsahuje sekvenční kód. Sekvenční kód může uživatel libovolně upravovat, vkládat si zde své fragmenty kódu a používat definované globální funkce a proměnné z hlavičkového kódu projektu. Tento hlavičkový kód se píše v nastavení projektu. Zvláštním případem přechodu je Kolektivní přechod. Je to typ, který neobsahuje sekvenční kód. Jeho úkolem je zajistit nějaký druh kolektivní komunikace mezi všemi procesy aplikace. Na obrázku 3.3 lze vidět editor sekvenčního kódu v nástroji Kaira.

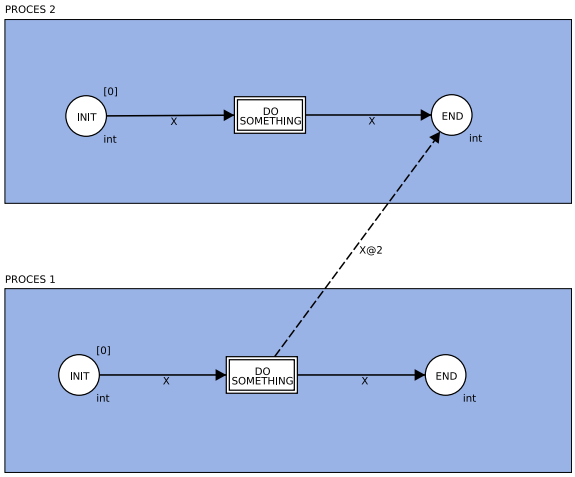
Pokud síť obsahuje více proveditelných přechodů současně, jsou zde dvě možnosti, jak se řeší jejich provedení. Pokud jednotlivé přechody nemaji úrčenou svojí prioritu provedení, není zaručena posloupnost provedení těchto přechodů a jsou libovolně prováděny a mohou být prováděny i současně. Pokud ale přechod má úrčenou prioritu, funguje zde provádění jako v prioritní frontě. Přechody s větší prioritou jsou provedeny dříve, než přechody s nižší prioritou.

Kromě přechodů mohou obsahovat sekvenční kód také místa. Pro místa je tento sekvenční kód pouze inicializační. To znamená, že při spuštění simulace se naplní místo hodnotami, spustí se inicializační kód, a poté už se v simulaci kód znovu neprovádí. Oproti tomu sice hrany obsahují pouhé fragmenty kódu, ty se ale provádí při každém průchodu tokenu hranou. Mohou to být jakési podmínky průchodu. Lehce rozdílná syn-

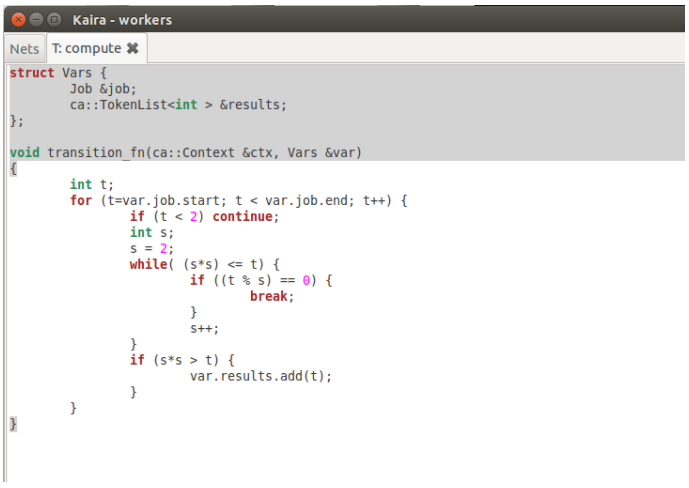
[5] http://www.open-mpi.org/
[6] http://www.mcs.anl.gov/research/projects/mpich2/
[7] http://www.lam-mpi.org/
[8] http://www.mpi-forum.org/
[9] http://verif.cs.vsb.cz/kaira/
[10] http://www.springer.com/us/book/9783642002830

taxe fragmentu kódu na hranách, od C++ syntaxe, je popsána v disertační práci autora nástroje Kaira [1]

Pokud hrana obsahuje pouze název proměnné, kterou předává, nevzniká komunikace mezi jednotlivými procesy. Pokud ale hrana obsahuje fragment kódu, který by značil, že předává data na jiný proces (kód směřující proměnou na jiný proces má tvar `název_proměnné@číslo_procesu`), funguje zde komunikace jako na obrázku 3.2, kde tyto přechody mezi procesy značí čárkovaná šipka. Každý proces vlastní svoji vlastní kopii sítě.



Obrázek 3.2: Ukázka komunikace mezi jednotlivými procesy pomocí barevných Petriho sítí v nástroji Kaira.



Obrázek 3.3: Editor sekvenčního kódu přechodu v nástroji Kaira.

4 Podpora testování sekvenčního kódu

Hlavním nedostatkem Kairy byla nemožnost testování sekvenčního kódu uvnitř přechodu. Dříve to bylo možné pouze za pomocí externích nástrojů. Zároveň zde byl reálný problém se sekvenčním kódem psaným v C/C++. Jelikož jsou zde typicky použity pouhé fragmenty kódu, které jsou napojeny na vizuální část programu, není tak možné snadno je přenést do externího nástroje. Hlavním úkolem je tedy umožnit uživateli vygenerovat si vlastní samostatnou síť pro konkrétní přechod a z této sítě poté nechat uživatele vytvořit si test. Tato síť bude nezávislá na zbytku programu a bude ji dále možno jednoduše převést do externího vývojového prostředí, a to vygenerováním C/C++ kódu nástrojem Kaira. V externím vývojovém prostředí lze poté generovaný kód např. debugovat.

Zároveň je potřeba vytvořit funkcionalitu, která by umožnila testovat daný fragment kódu aplikace v samotné Kaiře tak, aby měl uživatel připraveny veškeré testovací funkce, které by potřeboval pro porovnávání hodnot. Mohl by si dále sekvenční kód upravit dle svých požadavků. To by mělo zajistit možnost testovat kód přímo v nástroji Kaira bez potřeby dále ho generovat a převádět do externího vývojového prostředí.

4.1 Možnosti řešení

Je více možností jak implementovat funkcionalitu, která řeší tento úkol. První možností je upravit vygenerovaný kód sítě tak, aby obsahoval pouze funkčnost přechodu a jeho okolí. Další možností řešení problému je generování nových testovacích sítí, které vytváří novou síť ze sítě již existující, a to tak, že se odeberou komponenty, které blízce nesouvisí s testovaným přechodem.

4.2 Úprava vygenerovaného kódu

Při sestavení a simulaci navržené sítě uživatelem se generuje výsledný C/C++ kód, který reprezentuje aplikaci navrženou v Kaiře. Tento kód se generuje automaticky v nástroji. Toho by se dalo využít při tvorbě testovací sítě tak, že by se generoval pouze kód, který blízce souvisí s daným přechodem, tedy přechodová funkce přechodu a funkce vstupních a výstupních míst a hran. Daný kód by potom obsahoval jen to potřebné pro testování sekvenčního kódu a tím pádem by byl obsahově menší, byl by uživateli více srozumitelný a kód by představoval pouze okamžik, kdy se uživatel nachází před a po provedení přechodu.

Výhodou tohoto řešení je zkrácený vygenerovaný kód, který je kratší než celý původní program. Nevýhodou tohoto řešení ale je, že daný kód se nedá opět vizualizovat v Kaiře a upravovat ho zde. Proto by bylo potřeba využít externí vývojové prostředí.

4.3 Generování nových podsítí

Podstatou tohoto řešení je dát možnosti jednoduššího exportu testu do externího nástroje generováním výsledné aplikace, které je obsahově menší, umožnující pracovat pouze s

testovaným přechodem a využitím testovacích funkcí assert, které jsou typické pro testovací nástroje z architektury xUnit. To vše v samotném nástroji Kaira.

Hlavní podstatou je vygenerování nové sítě do již stávajícího projektu, popřípadě do nového, kdy nově vygenerovaná síť bude mít pouze testovaný přechod a vstupní a výstupní místa daného přechodu. To umožní pracovat pouze s testovaným přechodem. Důležitou funkčností obou řešení je možnost uložit si hodnoty na vstupních místech přechodu před jeho provedením. Pro generování nových podsítí je ale tato funkčnost důležitější. Díky této funkčnosti lze pracovat s testovanou sítí jako se sítí celou, jen díky tomu, že vstupní místa si načtou uživatelem uložené hodnoty, které se při simulaci generované podsítě vizualizují v samotném nástroji Kaira, a které se poté po jednom posílají do přechodu jako vstupní hodnoty. Testovaná síť se díky tomu chová jako by se vše před tím a potom provedlo někde mimo a uživatel se může věnovat pouze tomu, aby si otestoval sekvenční kód testovaného přechodu. Osamostatnění přechodu a jeho okolí do nové sítě lze využít pro snadnější export do externího vývojového prostředí, protože výsledný vygenerovaný kód obsahuje jen funkčnost spojenou s přechodem.

Výhodou implementace tohoto řešení je, že Kaira nabízí generování a vizualizaci sítí. To umožní vygenerovat novou síť z předešlé umazáním komponent sítě a vizualizovat ji v Kaiře jako novou síť, se kterou lze dále pracovat, jako by to byla samotná aplikace vyvíjená v Kaiře. Další výhodou je zkrácený vygenerovaný kód, který je kratší než celý původní program.

4.4 Porovnání možností

Obě možnosti řešení zkracují vygenerovaný kód výsledné aplikace. Kód je díky tomu více srozumitelný a obsahuje pouze tu část sítě, která závisí na testovaném sekvenčním kódu. Možnost úpravy generovaného kódu ale neumožňuje následné úpravy sítě a testovaní v nástroji Kaira. Je tudíž potřeba pro následnou tvorbu testů a provádění testů použít externí nástroj. Oproti tomu, generování nových podsítí umožňuje úpravu vygenerované podsítě v samotném nástroji, vytvoření testu z podsítě a následné spouštění jednotlivých testů v testovacím prostředí nástroje Kaira.

5 Implementace nové funkcionality

Bylo vybráno řešení Generování nových testovacích sítí, kvůli již implementovanému generátoru sítí v samotném vývojovém prostředí, a díky tomu není pro samotné testování potřeba použití externích nástrojů. Řešením je nově vyvinutá funkcionalita, která dovoluje testovat sekvenční část libovolného nekolektivního přechodu v modelu aplikace v nástroji Kaira. Rozhraní navíc umožňuje ukládat aktuální hodnoty všech vstupních míst přechodu a zároveň využít definovaných funkcí `store_binding` a `load_binding`. Tyto funkce může uživatel použít pro ukládání hodnot a načítání uložených hodnot přímo v kódu. To znamená, že mu dává na výběr mezi použitím kontextového menu v simulaci a napsáním si této funkčnosti sám do sekvenčního kódu míst. Protože je potřeba zajistit konzistenci a aktuálnost uložených dat ze simulace, ukládá si každý projekt informace o tom, jaké testovací sítě z něj byly vytvořeny. Když se poté uloží hodnoty v simulaci, aktualizují se všechny data do všech vytvořených testovacích sítí, které byly generovány do nových projektů.

Nakonec umožňuje používat definovanou assert metodu, která vyhodnocuje a testuje očekávané hodnoty s aktuálními hodnotami v jednotlivých krocích simulace. Uživatel si může do této metody vložit vlastní definovanou porovnávací funkci ve tvaru `bool name(T first, T second)`, díky které dostává volnost v porovnávání objektů. Samozřejmě uživatelem definovaná funkce nemusí porovnávat, ale musí vracet True/False, podle kterého se poté vyhodnocuje, zda hodnota prošla nebo ne. Pokud se porovnávané objekty nerovnají nebo metoda vrací False, uživatel je upozorněn v konzoli na neúspěšnost testu a může být ukončen běh programu. Pokud se rovnají, běh aplikace pokračuje dále a uživatel si může zvolit, zda chce být upozorněn na úspěšné provedění testu.

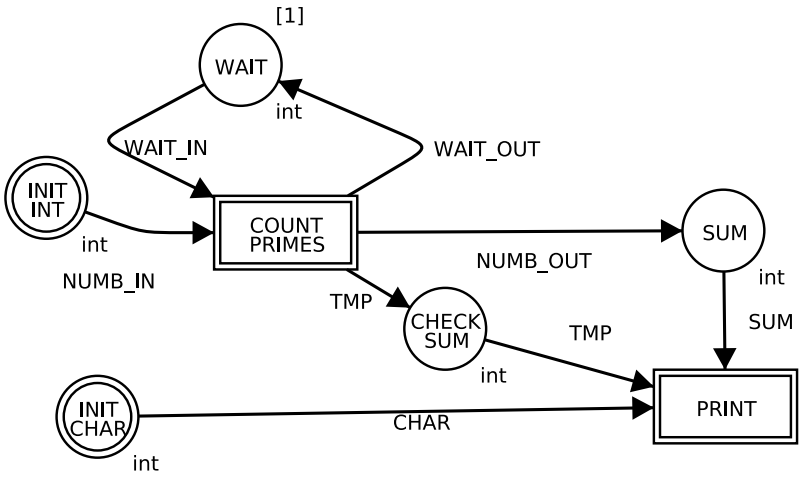
Kromě samotného tvoření testu, nabízí nová funkcionalita také testovací prostředí, které umožní uživateli spouštět sadu testů, upravovat jednotlivé sady testů a procházet výsledky jednotlivých testování.

5.1 Vytvoření nové sítě a její spuštění

Pro vytvoření nové sítě je potřeba vytvořit nový projekt. Nový projekt se vytvoří z menu *Project*. Po jeho vytvoření se objeví horní lišta s nabídkou komponent, které se mohou přidat do sítě. Také je zde možnost upravit nastavení projektu a jeho hlavičkový kód, který se ve výsledku chová jako globální. Tento kód se dá upravit v menu *Edit* pod *Edit head code*. Přidáním pár komponent a jejich úpravou lze vytvořit síť jako na obrázku 5.1. Tato síť má za úkol vypočíst počet prvočísel v úrčitém rozsahu, který vychází ze součtu proměnných *WAIT_IN* a *NUMB_IN*. Místo *WAIT* slouží k odložení hodnoty *NUMB_IN* na další provedení přechodu.

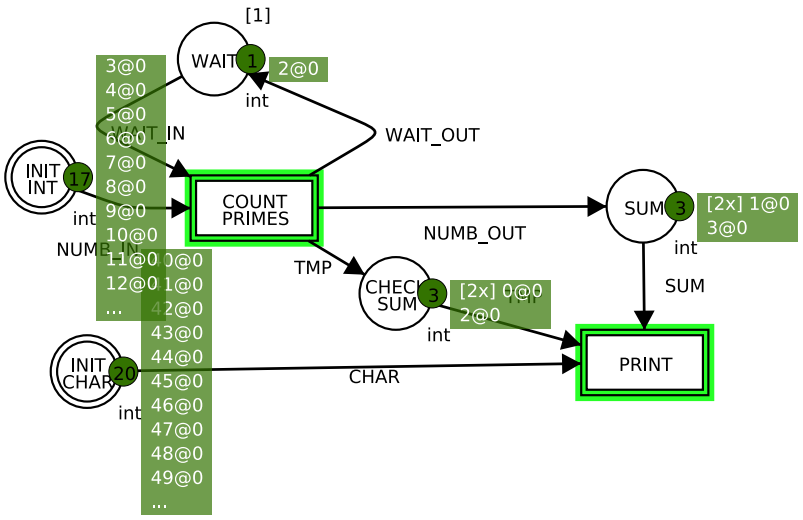
Vyvoláním kontextového menu přechodu lze editovat jeho sekvenční kód (viz *Edit code*). Pro editaci sekvenčního kódu se otevře vestavěný editor, viz obr: 3.3. Pokud přechod obsahuje sekvenční kód, je zobrazen jako obdelník s dvojitou čarou (viz přechod *COUNT PRIMES* na obr.: 5.1).

Existuje tedy vytvořená spustitelná síť. Volbou menu *Simulation - Run Simulation* se spustí simulace sítě. Pokud má projekt v nastavení přidané parametry, zobrazí se před



Obrázek 5.1: Ukázka libovolné síte v nástroji Kaira.

samotným spuštěním dialog s nastavením hodnot parametrů. Jinak se zobrazí dialog pouze s natavením počtu procesů, které bude síť využívat. Jak vypadá síť v simulaci lze vidět na obrázku 5.2.



Obrázek 5.2: Ukázka simulace libovolné síte v nástroji Kaira.

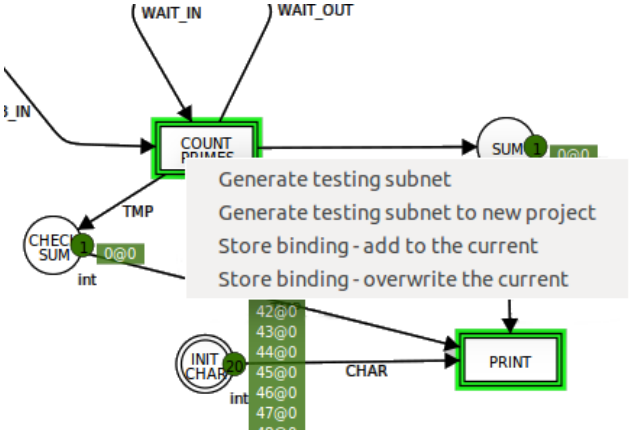
5.2 Počátek testování

Po dobu provádění přechodů v simulaci může uživatel narazit na hodnoty, které jsou podle něj neočekávané. Proto je dobré vygenerovat si síť, kde lze otestovat sekvenční kód přechodu. Tento kód lze poté upravovat nezávisle na zbytku sítě, nezmění se přitom celková funkčnost aplikace, dokud se opravený kód neumístí zpět do hlavní sítě. Tes-

tovací síť by měla v případě nutnosti generovat novou aplikaci, která bude obsahovat pouze kód potřebný pro spouštění testů pro testovaný přechod.

Pro vygenerování testovací sítě a pro ukládání hodnot, tedy ovládání všech implementovaných funkcí testování, slouží kontextové menu libovolného přechodu, které lze vidět na obrázku 5.3. Kontextové menu nabízí 4 možnosti, dvě možnosti generování testovací sítě a dvě možnosti ukládání hodnot, které se poté promítají do testovaných sítí jako inicializované hodnoty. Ukládání hodnot je ale možné pouze u přechodu, který je v danou chvíli proveditelný. Pro přechody, které jsou kolektivní není povolena žádná možnost pro testování. Je to z toho důvodu, že kolektivní přechod nemůže obsahovat sekvenční kód, tudíž nemá smysl testovat takovýto přechod.

Pokud uživatel nechce tvořit sítě pro test generováním sítí, nabízí se mu možnost vytvořit si takovou síť sám. Jediné čím se bude tvorba testovací sítě lišit je to, že si uživatel nebude generovat síť automaticky, ale upraví si síť tak, aby z ní byl test. Tedy odstraní nepotřebné komponenty sítě a dodá přechod *QUIT TRANSITION*, který poté korektně ukončí test. Může také využít rozhraní pro využití testovacích funkcí a poté svůj vlastní test přidat do testovacího prostředí. Nemusí si tak nechat vygenerovat síť generátorem a přesto smí použít funkčnosti nové funkcionality.



Obrázek 5.3: Kontextové menu proveditelného přechodu v nástroji Kaira.

5.3 Generování testovacích sítí

První volbou kontextového menu přechodu je *Generate testing subnet*. Po zvolení této možnosti se generuje nová testovací síť, která se poté nachází ve stejném projektu jako hlavní síť. Na obrázku 5.4 lze vidět nově vygenerovanou síť pro přechod s názvem *COUNT PRIMES* ze sítě na obrázku 5.1.

Generování testovací sítě probíhá tak, že se jako první znovu načte celá síť, která se poté zkopíruje do paměti. Nově zkopírovaná síť změní každé komponentě sítě id. Poté se prochází jednotlivé komponenty sítě a kontroluje se, zda blízce souvisí s testovaným přechodem. To znamená, že pokud je prvek vstupním nebo výstupním místem anebo

vstupní nebo výstupní hranou, tak poté tento prvek blízce souvisí s testovaným přechodem. Pokud se tedy při procházení sítě narazí na prvek, který blízce nesouvisí, tak se z testovací sítě odstraní.

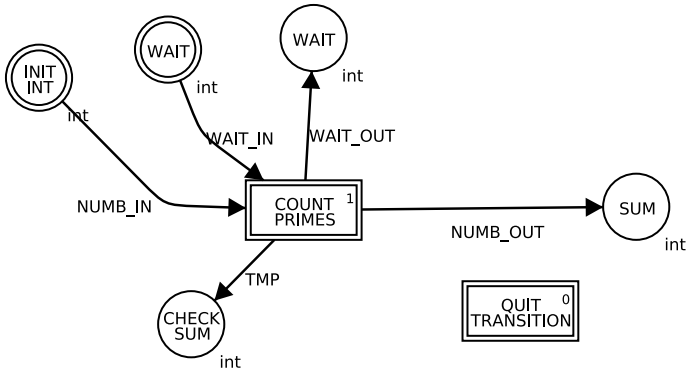
Kromě smazání některých komponent sítě se provádí několik dalších věcí, přidání nového přechodu, nastavení priorit a nastavení sekvenčního kódu vstupním místům. Všem vstupním místům se maže jejich sekvenční kód a vkládá se jim nový. A to `ca::load( place_data_file, place_id)`. Tato funkce zajišťuje načtení uložených hodnot ze simulace.

Aby byl proces provádění sítě vždy řádně ukončen, nastavuje se testovanému přechodu priorita 1 a přidává se do generováné sítě jeden nový přechod. Přechod se nazývá `QUIT TRANSITION` a má nastavenou prioritu 0. Tento přechod obsahuje sekvenční kód, který zaručí správné ukončení procesu sítě a tím kódem je `ctx.quit()`. Tento příkaz ukončí všechny vlákna, uvolní paměť a poté ukončí samotný proces sítě.

Generování testů je omezeno. Pokud byl test vygenerován do stejného projektu, je znemožněno generovat další testy pro stejný přechod v tom samém projektu. Přesto lze generovat další testy ze stejného přechodu do jiných projektů. Jakmile bude smazána síť, která obsahuje vygenerovaný test pro daný přechod, možnost generovat testy pro stejný přechod do stejného projektu se znovu povolí.

Zvláštním případem při generování testu je vstupně-výstupní místo. Příkladem tohoto místa je místo *WAIT* v ukázkové sítí na obr.: 5.1. Dochází zde totiž k rozdělení z jednoho místa na dvě. Ukázka rozdělení je vidět na obr.: 5.4. Důvodem rozdělení je to, že na vstupní a výstupní hraně mohou být různé výrazy, které se potřebují zachovat. Tedy vstupní hodnoty jsou vidět na vstupním místě, které je zároveň i inicializované. Výstupní hodnoty z přechodu lze vidět na výstupním místě. Nerozdělením místa by se také poté u něj hromadili výstupní tokeny přechodu, které by se vmíchaly do těch vstupních.

Dalším zvláštním případem, který navazuje na předešlý je dvojitá (*bidirectional*) vazba. Tato vazba zasílá tokeny na obě strany. Tato vazba ale zůstává nezměněna. Nerozpadá se tedy na dvě místa jako vstupně výstupní místo. Ukázka takové dvojité vazby je na obrázku 5.5. Tato vazba se nerozděluje, protože výraz na hranách je vždy stejný, proto není potřeba rozdělovat tuto vazbu.



Obrázek 5.4: Vygenerovaná testovací síť v nástroji Kaira.

Obrázek 5.5: Dvojitá vazba mezi místem a přechodem v nástroji Kaira.

5.4 Nový projekt

Volba *Generate testing subnet to new project* generuje testovací síť do nového projektu. Po zvolení umístění a názvu nového projektu se vygeneruje nová síť a otevře se nový projekt.

Po jeho vytvoření se nakopíruje celé nestavení projektu do testovacího projektu. Mezi nastavení, které se kopíruje, patří hlavičkový kód, parametry projektu a knihovny, které jsou přidány do seznamu souborů v nastavení projektu (viz *Edit project config - Build*). Kopírování knihoven je ale omezeno pouze na ty, které uživatel naincluduje do hlavičkového kódu projektu. Zjištění všech takto nalinkovaných knihoven se provádí parsováním kódu a porovnává se každý naincludovaný soubor se seznamem souborů přidaných v nastavení. Pokud nastane shoda, kopíruje se hlavičkový i zdrojový soubor. Pokud se parsováním nalezne soubor, který není v seznamu, soubory se nekopírují. Pokud poté nelze spustit simulaci testu z důvodu chybějící knihovny, kterou neměl uživatel správně přidanou do projektu, nahlásí Kaira do konzole informace o chybějících knihovnách. Důvodem takovéhoto jednání je to, že pro danou funkčnost není potřeba parsovat každý hlavičkový a zdrojový soubor použitý v projektu a zjišťovat všechny includované knihovny. Pokud by měl uživatel několik desítek až stovek includovaných knihoven, které by se např. navzájem cyklily, mohlo by to být časově velmi náročné.

Pro zajištění aktuálnosti ukládaných dat ve všech testovacích projektech se při ukládání hodnot v simulaci kopírují ukládané data do všech takto vytvořených projektů.

Bylo v plánu generování testovací sítě do nového projektu, aniž by se narušila simulace hlavní sítě, a aniž by se muselo přepnout na nově vytvořený projekt. Bohužel zde nastaly problémy ohledně událostí, které se volají v různých situacích a zajistí např. to, že se projekt po změně uloží, nebo že se promítne nová síť v projektu po jejím přidání. Tyto eventy ale potřebují pro svůj chod, aby byl projekt nastaven jako hlavní v aplikaci. Z tohoto důvodu se při generování do nového projektu přepne na nový projekt a nezůstává tedy spuštěn ten s hlavní sítí.

5.5 Ukládání hodnot

Každá hodnota v určitém místě je reprezentována jako *Token*. Místa si udržují aktuální tokeny v poli, které je funkční reprezentací fronty (první dovnitř, první ven). Každý token v sobě udržuje hodnotu podle typu, který má místo přiřazeno. Kromě vlastní hodnoty, nebo objektu, udržuje také informaci o tom, na kterém procesu se nachází.

Uživatel má možnost tyto tokeny ukládat. Vybráním příkazu *STORE BINDING - add to the current*, z kontextového menu na obr.: 5.3, se uloží aktuální hodnoty ze všech vstup-

ních míst. Uloží se ale pouze ty hodnoty, které jsou právě na řadě a lze je uložit pouze u přechodu, který je aktuálně proveditelný. Ukládané hodnoty se připíšou k těm, které již byly uloženy.

Vybráním příkazu *STORE BINDING - overwrite the current* se ukládají hodnoty stejně jako u předchozího příkladu, ale s tím rozdílem, že se hodnoty nepřidávají, ale přepíšou všechny již dříve uložené.

Pokud chce uživatel ukládat tokeny na vstupních místech přechodu, musí být daný přechod proveditelný. Další podmínkou je, aby si uživatel určil výběrem v levé nabídce, ze kterého procesu se budou data ukládat. Pokud si uživatel vybere možnost zobrazení dat ze všech procesů, vstupní tokeny, které jsou aktuálně na řadě, se načítají z náhodného procesu a poté se ukládají.

Tokeny se ukládají do souborů, kde každý je pojmenován *id_vstupniho_mista.data* a jsou umístěny do složky, která je pojmenována *id_prechodu*. Tyto složky se pak kopírují do všech testovacích projektů při ukládání hodnot.

Důležitou věcí, pro ukládání a načítání složitějších typů, je mít správně definovaný packer a unpacker. Packerem se myslí funkce, která definuje zabalování objektů do serializovatelné podoby a unpacker je funkce, která definuje, jak vytvořit objekt ze serializované podoby. Packer a Unpacker mohou být definovany globálně v hlavičce projektu, nebo každý typ, třída, či struktura mohou definovat packer a unpacker vlastní. Ukázka packeru a unpackeru pro strukturu Job lze vidět na výpisu kódu 1.

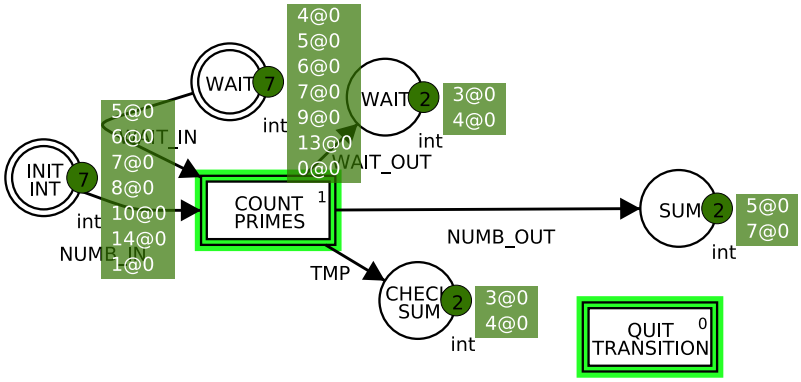
Pro načtení uložených hodnot do vstupních míst se používá metoda `ca::load()`. Tato metoda se automaticky vkládá do každého vstupního místa při generování testovací sítě. Metoda načítá všechny uložené hodnoty v pořadí, v jakém tam byly zapsány a inicializuje tím vstupní místo sítě. Pro dané hodnoty se neukládá na jakém procesu byly uloženy, tudíž se všechny data po načtení nachází na prvním možném procesu, tedy proces s id „0“. Důvod ukládaní tokenu bez informace na jakém procesu se nachází, je ten, že se testuje sekvenční kód, který je sériový, ne paralelní. Na obrázku 5.6 lze vidět ukázku simulace testovací sítě s načtenými uloženými hodnotami na vstupních místech testovaného přechodu. Simulaci celé sítě lze vidět na obrázku 5.2, kde jsou také vidět všechny inicializované hodnoty na vstupním místě. Tudíž je zde vidět rozdíl v hodnotách v simulaci.

```
struct Job {
    int  start ;
    int  end;

    void pack(ca::Packer &p) const {
        p << start  << end;
    }

    void unpack(ca::Unpacker &p) {
        p >> start  >> end;
    }
};
```

Výpis 1: Ukázka packeru a unpackeru pro strukturu Job.



Obrázek 5.6: Ukázka simulace testovací sítě s několika uloženými hodnotami.

5.5.1 Načítání hodnot uživatelem

Kromě automatického načítání hodnot, má uživatel možnost sám si načítat hodnoty na vstupních místech. Má pro to připravenou funkci. Načítá hodnoty do místa ze souboru definovaného uživatelem.

Daná funkce je již dříve zmíněna `void ca::load(const std::string &path, TokenList<T> &token_list)`. Používá se pro načítání hodnot ze souboru. Prvním parameterem je umístění souboru, který má uložené data pro vstupní místo. Druhým parametrem je samotný TokenList, který je druhým parametrem samotné funkce místa. Tato funkce načte do místa všechny uložené hodnoty v souboru, pokud jsou správného typu jako místo, do kterého se načítá.

Pokud by chtěl uživatel data ukládat v kódu, tak zatím tuto možnost nemá. Existuje sice metoda *ca::store*, bohužel se tato metoda volá pouze z generovaného kódu a dosazuje se tam objekt, který reprezentuje místo, což ze sekvenčního kódu v Kaiře není možné získat.

5.6 Rozhraní pro využití testovacích funkcí

Tvrzení (Assertion) je metoda, která ověřuje stav nebo chování jednotky v rámci testu. Selhání tvrzení obvykle indikuje výjimku a zastaví vykonávání aktuálního testu. V nové funkcionalitě má uživatel možnost využít nově definovanou metodu, která představuje požadovanou funkčnost.

Nová funkcionalita nabízí uživateli použití metody assertEquals. Tato metoda je definována se třemi nebo čtyřmi povinnými parametry a dvěma nepovinnými a je definována ve třídě *Context*. Metoda porovnává objekty základní porovnávací funkcí. Uživatel může definovat vlastní porovnávací funkci. Pokud se objekty nerovnají, systém může ukončit simulaci a poté vypíše do konzole zprávu ve tvaru *ASSERT EQUALS: USER_MSG - [Fail] Process: proccess_id (Expected: expected_value, Actual: actual_value)*. Pokud se porovnávané hodnoty rovnají, systém může vypsat zprávu o úspěchu ve tvaru *ASSERT EQUALS: USER_MSG - [Ok]*, nestane se nic a simulace pokračuje dále.

Pokud se použije metoda se třemi parametry, vkládají se pouze dva objekty, které se mají porovnávat a zpráva, která se vypíše při neúspěchu. Jako funkce pro porovnání dvou projektů se použije dříve definovana základní porovnávací funkce. Příklad použití metody *assertEquals* se třemi parametry lze vidět ve výpisu kódu 2.

Pokud uživatel použije metodu se čtyřmi parametry, vkládá jako první parametr svou definovanou porovnávací funkci. Porovnávací funkce musí vracet true nebo false, jinak řečeno návratový typ z C++ `bool`. Musí mít dva parametry stejného typu. Musí být tedy ve tvaru `bool func_name(T first_object, second_object)`. Poté funguje stejně jak metoda se 3-mi parametry.

První nepovinný parametr říká, zda se má proces spuštěného testu ukončit, když se při porovnání hodnoty nerovnají. Parametr je bez uživatelské změny definován jako false. To znamená, že se po neúspěchu proces neukončí a pokračuje v běhu dále, dokud neskončí provedením přechodu *QUIT TRANSITION*. Důvodem takového nastavení je to, že většinou chce uživatel vědět všechny úspěšné a neúspěšné hodnoty. Pokud ale je snaha ukončit proces po prvním neúspěchu, tedy pouze zjistit, zda projdou všechny porovnání bez chyby, stačí tento parametr nastavit na true. Ukázka použití volání s nastavením tohoto parametru je na výpisu kódu 2.

Druhý nepovinný parametr říká, zda se má vypisovat zpráva, pokud bylo porovnání úspěšné. Parametr je bez uživatelské změny definován jako true, to znamná že se vypisují i úspěšné zprávy. Důvodem tohoto nastavení je to, že ve výsledku testování se vypisuje počet úspěšných a počet neúspěšných porovnání, tady celková úspěšnost testu, která se počítá z uložených zpráv z načteného logu testu. Pokud uživatel definuje, že je vypisovat nechce, bude vždy počet úspěšných roven nule. Důležitou věcí při definování tohoto parametru uživatelem je ta, že C++ neumožňuje při volání funkce zadat název parametru a jeho hodnotu tak, aby nemusel definovat i ostatní nepovynné parametry. Takže je potřeba definovat i předešlý nepovinný parametr o ukončení testu po neúspěchu. Ukázka volání metody assertEqual při uživatelském definování je na výpisu kódu 2.

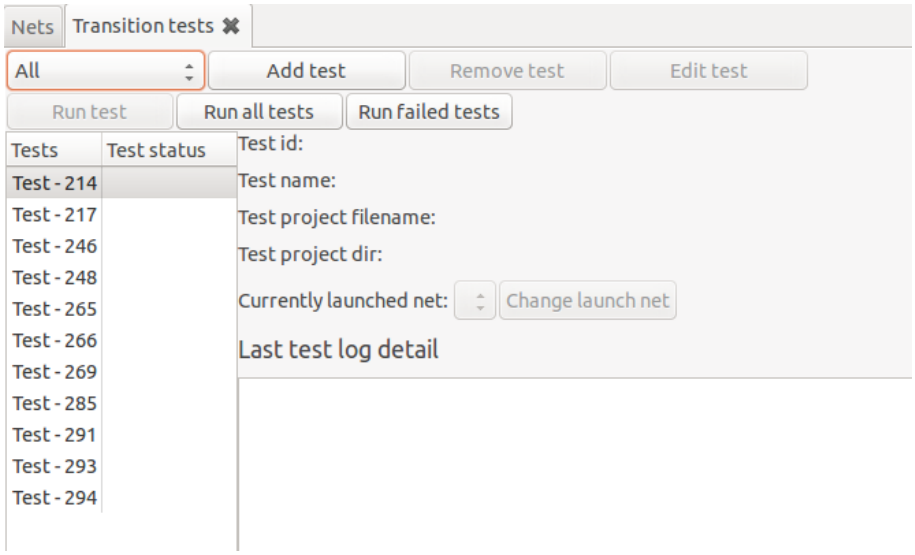
```
struct Vars {
    int &in;
    int &out;
};
void transition_fn (ca::Context &ctx, Vars &var)
{
    int expected = getExpectedValueFor(var.in);
    int counted = countSomething(var.in);
    ctx.assertEquals(expected, counted, "Expecte␣not␣equal␣counted␣in␣transition␣X");
    ctx.assertEquals(expected, counted, "Quit␣after␣not␣equality", true);
    ctx.assertEquals(expected, counted, "Dont␣show␣passed␣msg", false, true);
    var.out = counted;
}
```

Výpis 2: Použití metody assertEquals v sekvenčním kódu přechodu

5.7 Testovací prostředí

Nově vytvořená funkcionalita nabízí testovací prostředí, ve kterém dává uživateli možnost editovat jednotlivé testy, zobrazit detail testu s načtením logu výsledku daného testu, zobrazit a editovat seznam testů a nastavit danému testu síť, která se bude spouštět. Dále umožňuje spouštět jeden vybraný test, celou skupinu testů, nebo testy, které byly v posledním testování neúspěšné. Následně zobrazuje výsledky testování a statistiky jednotlivých porovnání v každém testu.

Pro zobrazení testovacího prostředí si uživatel vybere položku *Edit transition tests* v menu *Edit*. Po vybrání se otevře nový tab v Kaiře s názvem *Transition tests*, který obsahuje vše potřebné pro správu a spouštění testů. Na obrázku 5.7 je znázorněn vzhled testovacího prostředí.



Obrázek 5.7: Ukázka vzhledu testovacího prostředí nástroje Kaira.

5.7.1 Test

Nová funkcionalita nabízí ověření správnosti jednotlivých částí výsledné aplikace pomocí testů. Testem je zde myšlena síť projektu, kterou si uživatel upravil tak, aby ověřovala správnost určité části jeho aplikace. Takovou síť si může uživatel vytvořit sám, nebo může použít generátor podsítí z nově vytvořené funkcionality, která mu vytvoří podsíť, ale ne samotný test. Test si již z této podsítě uživatel musí vytvořit sám. Generátor má uživateli ulehčit pouze generování těchto testovacích sítí. S takto upravenou sítí může uživatel pracovat v testovacím prostředí. Takových testů může mít uživatel v jednom projektu více. Každému přidanému projektu s testy musí být nastavena síť, která se bude při testování spouštět.

Seznam testů je množina všech testů, která je vázana na projekt. Každý projekt může mít vlastní množinu těchto testů. Přidání, smazání a editace testů z této množiny jsou níže popsány.

Pokud má projekt v nastavení projektu zadány parametry, při simulaci sítě v nástroji Kaira se před spuštěním vyvolá dialog, kde uživatel může tyto parametry nastavit. Při generování testu se tyto parametry do testu kopírují. Při spouštění jednotlivých testů se poté nastavuje každému nekonstantnímu parametru jeho výchozí hodnota. Pokud by výchozí hodnotu neměl určenou, test by se nesputil, protože by neznal hodnoty všech potřebných parametrů. Proto je důležité, aby tyto parametry měli nastavenou výchozí hodnotu.

5.7.2 Zobrazení testů

Všechny přidané testy lze vidět v seznamu testů. Po výběru testu v daném seznamu se zobrazí detail testu. U každého testu může být zobrazen výsledek testování. Výsledek se zobrazuje pouze pokud byl test obsahem posledního testování. Jako výsledek testu může být zobrazena jedna ze tří možností. Výsledek *Passed* znamená, že při posledním spuštění testu proběhl test v pořádku a všechny porovnávané hodnoty se rovnaly. Výsledek *Failed* značí, že testování bylo neúspěšné. To znamená, že minimálně jednou se nerovnaly porovnávané hodnoty. Poslední možnost je *Build error*, která značí, že při generování a kompilaci testu z testovací sítě nastala chyba. Kde a jaká chyba nastala se vypíše v konzoli aplikace.

Uživatel má možnost filtrovat seznam testů podle výsledků testování. Má možnost zobrazit si pouze úspěšné, neúspěšné, které se nespustili díky chybě, všechny bezohledu na výsledek, nebo si může zobrazit všechny, které byly obsahem posledního testování. Po dokončení testování se automaticky mění filtr na poslední testované testy, aby uživatel viděl výsledky testování a nemusel je všechny hledat v seznamu.

5.7.3 Testovací log soubor

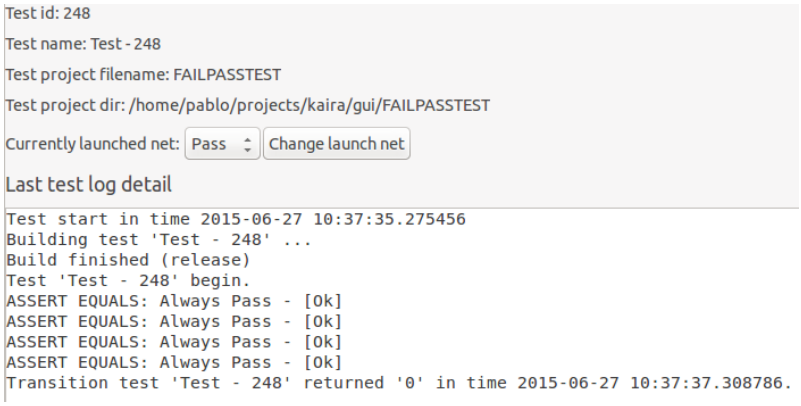
Testovací log soubor je textový dokument, který obsahuje informace o posledním spuštění testu. Obsahuje datum a čas, kdy byl test spuštěn, všechny výpisy testu (výpisy assert zprávy, ale také výpisy, které si uživatel definoval), datum a čas ukončení testu a nakonec jakou hodnotu test vrátil.

Logovací soubor pro každý test se nachází ve složce *Test logs* hlavního projektu a má stejné jméno jako samotný test. Uživatel si tedy může procházet jednotlivé testovací logy zvlášť, aniž by musel mít spuštěné testovací prostředí nástroje Kaira.

5.7.4 Detail testu

Po vybrání testu v seznamu testů se zobrazí jeho detail. Zobrazí se id a název testu, jeho umístění, název projektu, ve kterém se testovací síť nachází a aktuálně spouštěná síť při testování. Uživatel si tuto síť může měnit. Kromě podrobností o testu se taky načítá obsah testovacího logu. Testovací log obsahuje výsledek a výpis posledního provádění

testování daného testu. Detail vybraného testu s načteným logem testování lze vidět na obrázku 5.8.



Obrázek 5.8: Zobrazení detailu testu v testovacím prostředí nástroje Kaira.

5.7.5 Editace testu

Testovací prostředí umožňuje editovat samotný test otevřením testového projektu v nové instanci Kairy.

Button *Edit test*, který slouží pro editaci testu, se odemkne ve chvíli, kdy má uživatel vybrán test v seznamu testů. Po kliknutí se spustí nová instance nástroje Kaira, která má načtený projekt s vybraným testem. Uživatel si tedy může upravit test a poté ho uložit. Při dalším spuštění editovaného testu se již spouští test po úpravě.

5.7.6 Přidání testu

Existují dvě možnosti, jak přidat test do skupiny testů. Obě možnosti přidávají celý projekt, který vlastní daný test. Uživatel si pouze v detailu testu nastaví, která síť reprezentuje jemu požadovaný test.

První možnost přidání testu do seznamu testů je vygenerovaní podsítě ze simulace. Po vygenerování podsítě se vytvoří nová síť, která se poté musí upravit tak, aby se dala považovat za test. Takto neupravená síť je již ale přidána do seznamu testů hlavního projektu a lze už ji také spustit, protože při tomto způsobu přídání testu se tato síť nastaví jako spustitelná. Poté ji stačí upravit tak, aby se dala považovat za test. Tím je myšleno např. použití funkce *assertEquals*. Toto již ale musí uživatel napsat sám.

Druhou možností přidání testu je použití buttonu *Add test* v testovacím prostředí. Po kliknutí se zobrazí dialogové okno, kde se musí vybrat projekt, který obsahuje test. Po výběru se zobrazí přidaný test v seznamu a načtou se k němu informace v detailu testu. Pokud vybraný projekt vlastní více sítí, v detailu testu nabídne testovací prostředí výběr sítě, která reprezentuje požadovaný test. Pokud načtený projekt obsahuje více testů, může uživatel přidat tento projekt do seznamu testů vícekrát, a poté pouze u každého

testu nastaví, která síť se má spouštět. Po přidání testu tímto způsobem se nastaví, jako spouštěná síť, hlavní síť projektu.

5.7.7 Odebrání testu

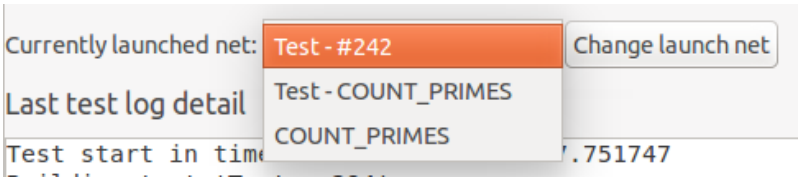
Pro odebrání testu slouží v testovacím prostředí button *Remove test*. Ten je odemknut ve chvíli, kdy uživatel vybere test ze seznamu testů. Odebrání testu nevymaže logovací soubor, ani samotný projekt s testem, pouze odstraní test ze seznamu testů.

5.7.8 Nastavení sítě pro testování

Projekt může obsahovat více sítí. Ne všechny tyto sítě mohou být testem. Při testování se spouští vždy jen jedna síť z daného projektu. Uživatel si může tuto síť vybrat. Pokud si uživatel síť nevybere, spouští se vždy ta, která je v seznamu sítí v projektu první.

Po zobrazeni detailu testu se zobrazí informace o projektu, ve kterém daný test je. Zobrazí se mu také nabídka, ve které jsou vypsány všechny sítě projektu (viz obrázek 5.9). Tyto sítě jsou v nabídce vypsány pod svým názvem, stačí tedy vybrat tu, která se jmenuje jako požadovaný test. Po výběru se musí potvrdit buttonem vedle, že uživatel souhlasí se změnou testovací sítě pro daný test. Po potvrzení se projeví změna a při dalším spuštění vybraného testu se už spouští test s novou sítí.

Důležitým faktorem u změny spouštěné sítě je to, aby názvy všech sítí v projektu byly jediněčné.



Obrázek 5.9: Výběr testovací sítě v testovacím prostředí nástroje Kaira.

5.7.9 Spouštění testů

Testovací prostředí nabízí tři možnosti spouštění testů. Spustit pouze jeden vybraný test ze seznamu testů, spustit všechny testy ze seznamu, nebo spustit testy, které byly neúspěšné z množiny posledních spuštěných testů.

Výběrem testu ze seznamu se odemkne button *Run test*, který spustí vybraný test. Po jeho spuštění se v konzoli vypíšou informace o kompilaci testu a poté se spustí. Po ukončení testu se zobrazí, u daného testu v seznamu, výsledek testování. Také se vypíše statistika úspěšných a neúspěšných porovnání v konzoli s tím, zda test uspěl nebo ne. Nakonec se aktualizuje detail testu, kdy se mění akorát testovací log, který je znovu načten a zobrazen.

Pro spuštění všech testů stačí kliknout na button *Run all tests*. Po kliknutí se začnou vypisovat do konzole informace o spuštění jednotlivých testů. Vždy se spustí jeden a

čeká se, dokud se neukončí. Po jeho ukončení se spoustí další v pořadí. Po ukončení posledního testu se vypíšou výsledné statistiky do konzole a aktualizuje se seznam testů tak, že se každému testu přiřadí jeho výsledek.

Pokud při posledním provádění testů existuje minimálně jeden test, který má výsledek testování Failed, může uživatel znova spustit daný test po kliknutí na button *Run failed tests*. Pokud existuje více takových testů, které jsou neúspěšné, spustí se všechny tyto testy znova.

Spouštění více testů naráz se provádí tak, že se všechny uloží do pole testů, kdy se vždy vezme z tohoto pole test, vygeneruje se výsledný C++ kód z testu, který se následně zkompiluje, a pokud se při kompilaci nenajde žádná chyba, test se spustí. Důležitou roli při spouštění testů zde hraje přechod označený jako *QUIT TRANSITION* (viz obrázek 5.4). Daný přechod má nastavenou nulovou prioritu a zajistí řádné ukončení testu po provedení všech přechodů v síti. Bez tohoto přechodu by se test nemusel správně ukončit, nebo by se neukončil vůbec a uživatel by ho musel vypínat zvlášť ve spuštěných procesech systému.

Důležitou věcí pro spuštění projektu, který obsahuje více sítí, je to, aby všechny sítě projektu byly kompilovatelné. Při kompilaci se generuje C++ kód z celého projektu, tedy i ze všech sítí, a pokud by jen jedna nebyla kompilovatelná, samotné spuštění testu skončí ve stavu *Build error*.

6 Testování sekvenčního kódu na hranách a v místech

Místa a hrany obsahují sekvenční kód, tak jako přechody. Testování sekvenčního kódu míst a hran je podobné jako je testování sekvenčího kódu přechodů. V přechodu se testování řeší tak, že se správně vygeneruje podsíť, která obsahuje pouze přechod a jeho vstupní a výstupní místa. Tyto komponenty jsou pro přechod důležité, aby se dal přechod otestovat a síť se dala spustit.

Místa mohou obsahovat pouze inicializační kód, který vkládá vstupní hodnoty do vstupních míst libovolného přechodu. Pro spuštění sítě je potřeba, aby mělo vstupní místo svou výstupní hranu a přechod. Proto u testování sekvenčního kódu míst postačí, aby se generovaly sítě, které zanechají jen testované místo, jeho výstupní hranu a přechod, na který hrana míří.

U hran je to v podstatě stejné. Vždy musí hrana vycházet z místa nebo z přechodu. Pokud bude vycházet z místa, na výstupu hrany bude přechod a do místa se nainicializují vstupní hodnoty, aby se mohl sekvenční kód hrany otestovat. Pokud bude hrana mířít z přechodu, přechod nemá jak vzít vstupní hodnoty. Proto se musí zachovat vstupní místo a hrana přechodu, poté testovaná hrana a výstupní místo hrany. Následně se na vstupní míso přechodu inicializují hodnoty.

Protože zde již existuje možnost ukládat si hodnoty z míst v simulaci, zbývá pouze doprogramovat generování takovýchto sítí, aby se z nich dal otestovat sekvenční kód hrany a nebo místa.

7 Zhodnocení a Závěr

Nově vytvořená funkcionalita dodává uživateli možnost, jednoduše si vygenerovat testovací podsítě, které si uživatel poté přetvoří na test. Tyto testovací sítě se generují pomocí základního generátoru sítí v nástroji Kaira. V síti jsou odstraněny nepotřebné komponenty a doplní se potřebné části kódu do vstupních míst pro to, aby byla síť spustitelná. Testovací sítě lze generovat do stejného projektu, ze kterého uživatel tvoří testy, nebo do projektu nového. Všechny takto vytvořené projekty mají aktuální ukládaná data. Pro uživatele zajímavé vstupní data přechodu, si uživatel ukládá v simulaci. Tím, že se generují menší testovací sítě, je také zajištěno, že výsledná aplikace, která se vytvoří z testované sítě, obsahuje pouze C/C++ kód týkající se přechodu a samotného projektu, jako je např. hlavičkový kód. Z tohoto důvodu je obsah kódu v aplikaci daleko menší a neobsahuje zbytečný kód, který může tuto aplikaci činit nepřehlednou. Tato výsledná aplikace uživateli umožní lokálně pracovat se sekvenčním kódem přechodu, aniž by musel reflektovat další část původní sítě, která může být poměrně komplexní. Pomocí této části kódu je uživatel schopen simulovat a optimalizovat různé stavy přechodu nezávisle na původní síti. Navíc v této práci přidávám možnost použití metody AssertEqual pro porovnání hodnot, které za běhu dokáží porovnat očekávané hodnoty s hodnotami skutečnými.

Dalším přínosem mé práce je testovací prostředí, které umožňuje vytvářet a spravovat množinu testů vázaných na projekt, spouštět tuto skupinu testů, jeden test, nebo pouze neúspěšnou část posledního testování, upravovat ji a zkoumat výsledky jednotlivých testů. Testovací prostředí také nabízí výpis všech těchto testů v seznamu, který se dá filtrovat podle výsledku jednotlivých testů.

Celkově je zřejmé, že mnou nově vytvořené funkcionality jsou dílčím, ale velmi důležitým přínosem pro rozšíření použitelnosti nástroje Kaira.

V průběhu práce jsem nalezl řadu dalších možností, jak nástroj Kaira rozšířit směrem, kterým se ubírá tato práce. Tím myslím dotáhnout testování do takového stavu, aby se braly v potaz samotné procesy, nasazení debugeru anebo dodat do samotného nástroje další prvky architektury xUnit. Testování kódu na hranách a v místech je s touto funkcionalitou jednodušší. Stačilo by přepsat algoritmus generování testovací sítě tak, aby vždy obsahovala jen to, co je potřeba pro komponentu, kterou chceme testovat.

7.1 Možnosti rozšíření

V nynější době jsou hodně rozšířené testovací frameworky, které se zakládají na architektuře xUnit. Tato architektura má některé specifické vlastnosti, např. spouštění všech testů v souboru najednou z vývojového prostředí, vyhodnocování výsledků, spouštění jen nesplněných testů a další vlastnosti, které jsou v poslední době neocenitelné při testování v době vývoje. Nově vytvořená funkcionalita některé tyto prvky nemá. Proto by se dalo dále pokračovat v tom, aby se v Kaiře vytvořila funkčnost, která všechny chybějící prvky architektury dodá. Mnou vytvořená funkcionalita je na to z části připravena. Projekt si pamatuje všechny jeho vytvořené testy, a dokáže také spouštět jednotlivé testy, nebo množinu těchto testů a zobrazovat výsledky v testovacím prostředí. Tudíž by stačilo implementovat další znaky xUnit architektury, které by byly potřeba.

Dalším rozšířením, které by pomohlo při vývoji v nástroji Kaira je nasazení debuggeru. Protože nelze ladit celou výslednou aplikaci v Kaiře, jednalo by se o možnost debuggovat pouze sekvenční kód jednotlivých přechodů a míst.

Tato práce se zabývá funkcionalitou, která umožňuje testovat sekvenční kód. Sekvenčním kódem se chápe kód, který se nachází uvnitř přechodu, a tudíž zde procesy nehrají žádnou roli. Proto není potřeba znát a pamatovat si id procesu, z kterého jsou ukládány hodnoty do testu. Pokud uživatelský kód pracuje s procesy, není v současnosti tato funkcionalita v rámci testování podporována. Do budoucna by se dalo vyřešit to, aby se ukládaly taky informace o procesu, z kterého hodnoty pochází a poté to dál zpracovávat tak, aby byl kód, který nějak zpracovává informace o procesu funkční.

Při generování testovací sítě do nového projektu se Kaira přepne na nový projekt. To zapříčiní vypnutí simulace a změnu projektu. Problém je zde u událostí, které jsou registrovány při změně sítě. Tyto události totiž obsluhuje samotná Kaira. Pokud tedy pracuji v paměti s jiným projektem, který není napojen na tyto události (není nastaveným výchozím projektem nástroje Kaira), je problém do něj ukládat síť a dále ji upravovat. Proto by bylo potřeba vymyslet napojení událostí tak, aby nebylo potřeba přepínat projekt na testovací při jeho generování.

8 Terminologický slovník

Termín	**Vysvětlení**
Binding	přiřazení hodnoty proměnným
Store-binding	Uložení všech vstupních hodnot daného přechodu
Provedení přechodu	Přechod zpracuje všechny vstupní hodnoty pomocí vnitřní funkce (fire-code) a následně předá výstupní hodnoty
Fire-code	Kód v přechodu, který se spustí pokaždé, kdy se provede přechod
Sémantika	Význam
Debugger	Program, pomocí kterého lze krokovat zdrojový kód programu. Umožní zastavit běh programu na libovolném míst ve zdrojovém kódu.
Debugging	Ladění programu pomocí debuggeru.
Breakpoint	Zarážka, na které debugger pozastaví běh programu.
Include	Přiložení hlavičkového souboru k zdrojovému.
Parsování	Proces analýzy posloupnosti formálních prvků s cílem určit jejich gramatickou strukturu vůči předem dané formální gramatice.
Token	Reprezentace hodnoty v daném místě.
Framework	Softwarová struktura, která slouží jako podpora při vývoji. Může obsahovat knihovny, rozhraní a další produkty, které jakkoli napomáhají ve vývoji softwaru.
Inicializace vstupního místa	Načtení tokenů do vstupního místa.
Button	Tlačítko, které po kliknutí vykoná určitou akci

9 Reference

[1] BÖHM, Stanislav. *Unifying Framework For Development of Message-Passing Applications* [online]. Ostrava, 2013 [cit. 2015-03-18]. Dostupné z: http://verif.cs.vsb.cz/sb/thesis.pdf. Ph.D. Thesis. VŠB - Technical University of Ostrava.

[2] Philip Koopman's Home Page. 1999. PAN, Jiantao. *Software Testing* [online]. Carnegie Mellon University [cit. 2015-05-06]. Dostupné z: http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

[3] PATTON, Ron. *Testování softwaru*. Praha: Computer Press, 2002. ISBN 80-7226-636-5.

[4] HLAVA, Tomáš. Funkční a nefunkční testy. In: *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování.* [online]. 2011 [cit. 2015-03-22]. Dostupné z: http://testovanisoftwaru.cz/tag/funkcni-testy/

[5] MARTINEK, David. Ladění a testování programů. In: *Jak na projekty v jazyce C* [online]. 2012 [cit. 2015-03-22]. Dostupné z: http://www.fit.vutbr.cz/~martinek/clang/debug.html

[6] Lawrence Livermore National Laboratory: High Performance Computing. 2014. BARNEY, Blaise. *Introduction to Parallel Computing* [online]. [cit. 2015-05-06]. Dostupné z: https://computing.llnl.gov/tutorials/parallel_comp/

[7] SILBERSCHATZ, Abraham, Peter B GALVIN a Greg GAGNE. *Operating system concepts with Java.* 6th ed. Hoboken: John Wiley & Sons, c2004, xxiii, 952 s. ISBN 0471489050.

[8] Introduction to Formal Verification. 1996. *The Donald O. Pederson Center for Electronic Systems Design* [online]. [cit. 2015-05-06]. Dostupné z: http://embedded.eecs.berkeley.edu/research/vis/doc/VisUser/vis_user/node4.html

[9] FOWLER, Martin. Xunit. In: *Martin Fowler* [online]. [cit. 2006-01-17]. Dostupné z: http://www.martinfowler.com/bliki/Xunit.html

[10] HAMILL, Paul. *Unit test frameworks* [online]. 1st ed. Sebastopol, CA: O'Reilly, 2004, xii, 198 p. [cit. 2015-04-21]. ISBN 978-0-596-10482-5.

[11] FOSTER, Ian. *Designing and building parallel programs: concepts and tools for parallel software engineering.* Reading: Addison-Wesley Publishing Company, c1995, xiii, 381 s., [8] s. příl. ISBN 0201575949.